Learning Probabilistic Models for Static Analysis Alarms

Hyunsu Kim
KAIST
Korea
hyunsu.kim00@kaist.ac.kr

Mukund Raghothaman
University of Southern California
USA
raghotha@usc.edu

Kihong Heo
KAIST
Korea
kihong.heo@kaist.ac.kr

ABSTRACT

We present BayeSmith, a general framework for automatically learning probabilistic models of static analysis alarms. Several probabilistic reasoning techniques have recently been proposed which incorporate external feedback on semantic facts and thereby reduce the user's alarm inspection burden. However, these approaches are fundamentally limited to models with pre-defined structure, and are therefore unable to learn or transfer knowledge regarding an analysis from one program to another. Furthermore, these probabilistic models often aggressively generalize from external feedback and falsely suppress real bugs. To address these problems, we propose BayeSmith that learns the structure and weights of the probabilistic model. Starting from an initial model and a set of training programs with bug labels, BayeSmith refines the model to effectively prioritize real bugs based on feedback. We evaluate the approach with two static analyses on a suite of C programs. We demonstrate that the learned models significantly improve the performance of three state-of-the-art probabilistic reasoning systems.

1 INTRODUCTION

To deal with the challenges of accuracy and alarm relevance, various probabilistic program reasoning mechanisms have been proposed for static program analyzers. Such systems initially report a set of alarms in the target program using the underlying analysis and compute the probability of each alarm based on a probabilistic model. Then, they prioritize static analysis alarms by incorporating external feedback on semantic facts from various sources such as the users [23, 39, 50], the old version of the program [15], or dynamic analysis results [5]. Upon receiving a response, they generalize from the feedback and prioritize the remaining alarms depending on their relevance to those inspected by the user. By rapidly focusing attention on the real bugs in the target program, these systems achieve a dramatic improvement in the usability of the static analyzer.

Despite their experimental success, much of this previous research has focused on the problem of *inference*, rather than on *learning*. Existing approaches based on probabilistic reasoning such as alarm ranking [5, 15, 39] only learn limited forms of transferable knowledge—such as the assignment of *weights* to the underlying probabilistic model—using standard methods such as the expectation-maximization algorithm [20]. In our observation, however, the capability of learning is fundamentally limited to the underlying *structure* of the probabilistic model.

In this paper, we propose a general framework for learning probabilistic models for static analysis alarms, that is applicable to various probabilistic reasoning systems [5, 15, 39]. The underlying systems initially construct Bayesian networks from the derivation structure of the alarms, and prioritize alarms using the induced confidence values. This ranking is repeatedly updated as the user inspects alarms and reports their findings. Ideally, these responses should always improve the confidence scores of true bugs and decrease those of false alarms. In practice, however, because of approximations caused by the underlying abstraction and during model recovery, they often incorrectly prioritize false alarms over true bugs. Our goal is to improve the accuracy of probabilistic models and mitigate the impact of these *false generalization* events. Given a set of training programs with bug labels, we learn logical rules that produce accurate Bayesian network models to reduce the number of user interactions until discovering all true alarms.

Notice that our problem (i.e., learning) is fundamentally different from that (i.e., inference) addressed in previous papers [5, 15, 39]. Learning and inference are complementary problems in AI/ML research, especially for Bayesian networks. The existing ones solely focused on the inference of Bayesian probabilistic models generated by a fixed hand-written set of rules. Instead, we shed light on the capability of learning the Bayesian models that can significantly improve the performance in multiple instances.

Our approach is based on two key ideas: (1) feedback-directed and (2) syntax-guided refinement. We first construct the Bayesian networks with an initial set of rules and evaluate the quality of interactive alarm prioritization using a given labeled data set. By observing the results, we capture the moments when a response to one alarm falsely generalizes to other alarms and degrades the overall quality of rankings. For such cases, our learning algorithm refines the rules that generate the inaccurate part of the Bayesian network. The refinement is guided by program syntax and encodes more detailed context of the labeled alarm to the rules by adding syntactic features which are directly derived from the grammar of the target language.

We have instantiated this approach in a tool named BayeSmith, and evaluate its effectiveness on a suite of widely used C programs, each comprising 5–112 KLOC with two analyses for C: an interval analysis for buffer overrun errors and a taint analysis for format-string and integer-overflow errors. We measure the effectiveness of BayeSmith with three state-of-the-art probabilistic program reasoning systems, each of which incorporates feedback from the users [39], the previous version of the program [15], and dynamic analysis results [5]. The learned rules by BayeSmith significantly

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510098

improve the performance of the existing systems with manually designed models by 30.7%, 54.3%, and 20.3%, respectively.

Contributions. This paper makes the following contributions:

(1) We present a framework for learning probabilistic models for static analysis alarms from data.
(2) We propose a feedback-directed algorithm that learns probabilistic models by mitigating false generalization events.
(3) We evaluate our approach with two state-of-the-art static analyzers and demonstrate significant improvements in three probabilistic program reasoning systems on a suite of C programs.

2 OVERVIEW

We illustrate our approach using the C program excerpted from wget-1.12 in Figure 1. Given the name of file (`file`), the function `ftp_parse_vms_ls` reads a line from the file (line 3) and tokenizes the line (line 4). Then, the function iterates over a token backward (line 6) and removes all digits after a special character `'^'` by adding `'\0'` (line 11). However, a buffer underflow bug can occur when all the elements in a token are digits. Pointer variable `p` may point to the starting position of such a token, so that the memory access to `p-1` at line 10 is potentially unsafe.

```
1  void ftp_parse_vms_ls(char *file) {
2      FILE *fp = fopen(file, 'r');
3      char *line = read_line (fp);
4      char *tok = strtok(line, "␣");
5      char *p = tok + strlen(tok);
6      while(p > tok) {
7          if(!c_isdigit(*p)) break; // false alarm #1
8          p--;
9      }
10     if (*(p - 1) != '^')          // true alarm (buffer underflow)
11         *p = '\0';                // false alarm #2
12 }
```

Figure 1: An example code excerpted from wget-1.12.

The buffer over/underflow analysis in Sparrow [37] detects the bug but also reports false alarms at the other buffer access expressions in the function. Sparrow's sound interval analysis can estimate that the memory access at line 10 can underflow the buffer. On the other hand, it cannot precisely capture the fact that the pointer `p` always points to a valid memory region even after the loop, because the contents of the string `tok` are determined at runtime so that the loop terminating condition becomes complicated.

Once such a set of alarms is generated, Sparrow ranks the alarms using a Bayesian alarm ranking system Bingo [15, 39]. Bingo computes a confidence score for each alarm and provides a ranking to the user. Then the user labels the top-ranked alarm so that Bingo performs Bayesian inference to update the scores of the remaining alarms. This probabilistic inference enables the user to interactively reason about the correctness of the program and effectively filters out false alarms while highlighting true alarms.

Although the alarm ranking system significantly reduces the alarm inspection burden of users, it often misinterprets user's response. For wget-1.12, Sparrow reports 891 alarms in total. After 166 user interactions with Bingo, the false alarm at line 7 is ranked at the top and the true alarm is also highly ranked (9th) in Figure 2(a). Once the user labels the top-ranked alarm as false, Bingo generalizes the feedback, thereby lowering the ranking of other correlated false alarm such as the alarm at line 11. However, this feedback also undesirably drops the rank of true alarm at line 10 from rank 9 to 462. Because the true alarm is also closely related to the labeled false alarm, Bingo falsely generalizes the user's label so that degrades the ranking. A similar issue again happens after 187 user interactions. The ranking of the true alarm drops from 2 to 505 after the user's labeling on the false alarm at line 11.

Our goal is to learn an effective probabilistic model based on Bayesian network that mitigates such *false generalizations*. We observed that false generalizations can happen because the structure of the Bayesian network is too coarse to capture the complex behavior of programs. In the original Bingo, this network is derived from a set of simple hand-written rules that approximates the abstract semantics of the underlying analyzer using def-use relations between program points [15]. The rules are general enough to capture the behavior of a large class of static analyses, but insufficient to differentiate various contexts. In the rest of this section, we illustrate why the false generalization occurs and how we learn more sophisticated rules from data.

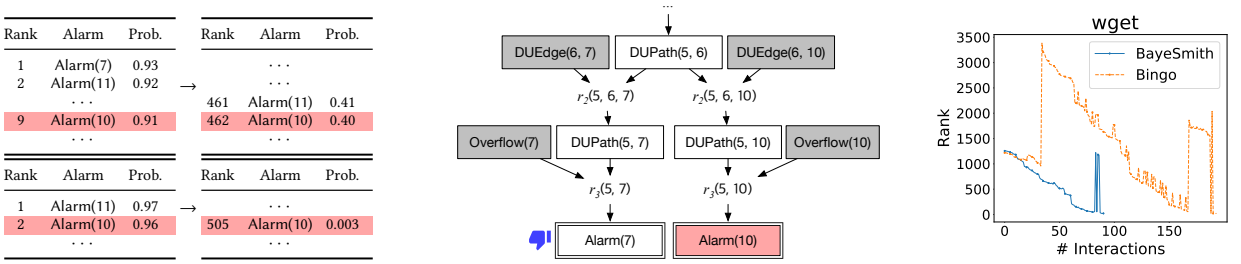
2.1 Approximating Static Analysis via Datalog

The buffer overflow analysis in Sparrow is based on the abstract interpretation framework [6] and uses a flow-, field-, and context-sensitive interval analysis [37]. While its actual abstract semantics involves sophisticated reasoning about numeric, pointer, and array values, it can be approximated by simple inference rules based on general def-use relations [36]. The inference rules written in Datalog are depicted in Figure 3 from which Bingo derives the Bayesian network structure. Notice that the approximated rules are used only for modeling portions of the whole reasoning process which are important for alarm ranking. They do not affect the behavior of the underlying analyzer written in arbitrary languages.

The Datalog program takes tuples in the input relations and derives tuples in the output relations using the set of rules. For example, input tuple $\mathsf{DUEdge}(6, 7)$ represents that there exists a direct data flow from the definition point of variable `p` at line 6 to its use point at line 7 in Figure 1.[1] Input tuple $\mathsf{Overflow}(7)$ approximates the detailed reasoning by the interval analysis and represents the fact that a buffer overflow may happen at line 7. The inference rules derive new output tuples from input and other output tuples. For example, rule r_2 derives an indirect data flow $\mathsf{DUPath}(c_1, c_3)$ from program point c_1 to c_3 using another data flows $\mathsf{DUPath}(c_1, c_2)$ and $\mathsf{DUEdge}(c_2, c_3)$. The ultimate goal of the analysis is to derive alarm tuples $\mathsf{Alarm}(c)$ that represent the fact that a potentially erroneous trace reaches program point c. We repeatedly apply the rules to all tuples until no more new output tuples are derived (i.e., fixed point).

Then we construct a derivation graph that shows all the reasoning steps to derive all alarms. An example derivation graph is shown in Figure 2(b) that explains how $\mathsf{Alarm}(7)$ and $\mathsf{Alarm}(10)$ are derived. For example, $\mathsf{Alarm}(7)$ is derived by applying r_3 to tuple $\mathsf{Overflow}(7)$ and $\mathsf{DUPath}(5, 7)$ which is derived by r_2 with $\mathsf{DUEdge}(6, 7)$ and $\mathsf{DUPath}(5, 6)$. Notice that the nodes with rule names mean grounded rules. For example, $r_2(5, 6, 7)$ describes rule

[1] We assume the SSA-style def-use relation. Thus, we consider the phi-node for variables `p` (i.e., line 6) at the loop head as a definition point.



(a) Ranking changes after 166 and 187 interactions **(b) Derivation graph** **(c) Performance comparison**

Figure 2: False generalization. Each data point in (c) represents the sum of the rankings of 6 true alarms.

Input relations	
$\mathsf{DUEdge}(c_1, c_2)$:	Immediate data flow from c_1 to c_2
$\mathsf{Overflow}(c)$:	Potential buffer overrun at c
Output relations	
$\mathsf{DUPath}(c_1, c_2)$:	Transitive data flow from c_1 to c_2
$\mathsf{Alarm}(c)$:	Potentially erroneous trace reaching c
Analysis rules	
$r_1 : \mathsf{DUPath}(c_1, c_2) \coloneq \mathsf{DUEdge}(c_1, c_2).$	
$r_2 : \mathsf{DUPath}(c_1, c_3) \coloneq \mathsf{DUPath}(c_1, c_2), \mathsf{DUEdge}(c_2, c_3).$	
$r_3 : \mathsf{Alarm}(c_2) \coloneq \mathsf{DUPath}(c_1, c_2), \mathsf{Overflow}(c_2).$	

Figure 3: Approximated interval analysis with simple inference rules. All variables indicate program points.

r_2 is triggered with tuples $\mathsf{DUPath}(5, 6)$ and $\mathsf{DUEdge}(6, 7)$, then derives conclusion $\mathsf{DUPath}(5, 7)$.

2.2 Bayesian Alarm Ranking System

Next, we convert the derivation graph into a Bayesian network following the scheme in the previous work [39]. The central idea is to quantify the incompleteness of each rule as a probability. Static analysis rules are typically designed to be sound but not always complete. This incompleteness is one of the main sources of false alarms. Suppose both $\mathsf{DUPath}(5, 6)$ and $\mathsf{DUEdge}(6, 7)$ are true in Figure 2(b). However, it does not necessarily mean the conclusion $\mathsf{DUPath}(6, 7)$ is always true in the concrete semantics, because complicated runtime behavior is approximated such as different loop iterations or nontrivial termination conditions. Bingo encodes this incompleteness using conditional probability. Each rule r is associated with a probability p_r. For example, the rules

$$\begin{aligned} \Pr\left(r_2(5,6,7) \mid \mathsf{DUPath}(5,6) \wedge \mathsf{DUEdge}(6,7)\right) &= p_{r_2} \\ \Pr\left(r_2(5,6,7) \mid \neg\mathsf{DUPath}(5,6) \vee \neg\mathsf{DUEdge}(6,7)\right) &= 0 \end{aligned}$$

represent the rule r_2 is successfully triggered with probability p_{r_2} if both $\mathsf{DUPath}(5, 6)$ and $\mathsf{DUEdge}(6, 7)$ are true, otherwise it is never triggered. Meanwhile, Bingo considers a conclusion of a rule is always true if the rule successfully fires. For example, $\Pr\left(\mathsf{DUPath}(5,7) \mid r_2(5,6,7)\right) = 1$. We use the default probability (0.99) of the Bingo system for rules and input tuples that do not have deriving rules.

Based on the Bayesian network constructed from the derivation graph, Bingo computes the probability of each alarm $\Pr\left(\mathsf{Alarm}(c) \mid \boldsymbol{e}\right)$ conditioned on the set of labels $\boldsymbol{e}$. The set $\boldsymbol{e}$ is initially empty and gradually includes more labels from the user. Every time the user provides a label for the top-ranked alarm, Bingo reranks the remaining alarms according to $\Pr\left(\mathsf{Alarm}(c) \mid \boldsymbol{e}\right)$.

2.3 The False Generalization Problem

Ideally, the user's labeling on a false alarm should only degrade the probability of other correlated false alarms in the network. Consider the alarm rankings before and after the user gives a negative label on $\mathsf{Alarm}(7)$ at the top of Figure 2(a). The labeling successfully generalizes to another false alarm $\mathsf{Alarm}(11)$ and degrades its probability from 0.92 to 0.41.

However, the negative feedback also falsely generalizes to the true alarm $\mathsf{Alarm}(10)$ and undesirably degrades its ranking from 9 to 462. The same effect also happens with another negative label on $\mathsf{Alarm}(11)$. In practice, such false generalizations hinder the user's effort to discover true alarms as early as possible.

We illustrate why this false generalization occurs with the derivation graph in Figure 2(b). Given a negative label on $\mathsf{Alarm}(7)$, Bingo computes the conditional probability $\Pr\left(\mathsf{Alarm}(10) \mid \neg\mathsf{Alarm}(7)\right)$ that indicates the probability of the true alarm after the negative labeling. According to the Bayesian network structure, $\mathsf{Alarm}(10)$ and $\mathsf{Alarm}(7)$ are conditionally independent given $\mathsf{DUPath}(5, 6)$. Therefore, the posterior probability is computed as follows:[2]:

$$\begin{aligned} &\Pr\left(\mathsf{Alarm}(10) \mid \neg\mathsf{Alarm}(7)\right) \qquad (1) \\ &= \Pr\left(\mathsf{Alarm}(10) \mid \mathsf{DUPath}(5,6)\right) \times \Pr\left(\mathsf{DUPath}(5,6) \mid \neg\mathsf{Alarm}(7)\right). \end{aligned}$$

The first term $\Pr\left(\mathsf{Alarm}(10) \mid \mathsf{DUPath}(5, 6)\right)$ evaluates to 0.96. For simplicity, we assume that the prior probability $\Pr\left(\mathsf{DUPath}(5, 6)\right) = 0.99$. Then, by Bayes' rule, the second term is evaluated as follows:

$$\Pr\left(\neg\mathsf{Alarm}(7) \mid \mathsf{DUPath}(5,6) \wedge \mathsf{DUEdge}(6,7)\right) \times 0.99^2 \times 0.07^{-1}$$

The probability $\Pr\left(\neg\mathsf{Alarm}(7) \mid \mathsf{DUPath}(5, 6) \wedge \mathsf{DUEdge}(6, 7)\right)$ is the sum of following disjoint cases:

- $r_2(5, 6, 7)$ misfired:

$$\begin{aligned} &\Pr\left(\neg\mathsf{Alarm}(7) \mid \neg r_2(5,6,7)\right) \times \Pr\left(\neg r_2(5,6,7) \mid \mathsf{DUPath}(5,6) \wedge \mathsf{DUEdge}(6,7)\right) \\ &= 1 \times (1 - p_{r_2}) = 0.01 \qquad (2) \end{aligned}$$

- $r_2(5, 6, 7)$ fired, $\mathsf{Overflow}(7)$ was true, but $r_3(5, 7)$ misfired:

$$\begin{aligned} &\Pr\left(\neg\mathsf{Alarm}(7) \mid r_2(5,6,7)\right) \times \Pr\left(r_2(5,6,7) \mid \mathsf{DUPath}(5,6) \wedge \mathsf{DUEdge}(6,7)\right) \\ &= (1 - p_{r_3}) \times 0.99^2 = 0.0098 \qquad (3) \end{aligned}$$

[2] All the details of the calculations in this subsection are described in Appendix ??.

- $r_2(5,6,7)$ fired but Overflow(7) was false:

$$\Pr\left(\neg\text{Alarm}(7) \mid r_2(5,6,7)\right) \times \Pr\left(r_2(5,6,7) \mid \text{DUPath}(5,6) \wedge \text{DUEdge}(6,7)\right) = 0.01 \times 0.99 = 0.0099 \quad (4)$$

Overall, $\Pr\left(\text{DUPath}(5,6) \mid \neg\text{Alarm}(7)\right)$ is evaluated to $0.0297 \times 0.99^2 \times 0.07^{-1} = 0.42$.

Finally, the true alarm now has a significantly low probability by multiplying the two terms: $\Pr\left(\text{Alarm}(10) \mid \neg\text{Alarm}(7)\right) = 0.96 \times 0.42 = 0.40$. Notice that the probability of the first term is high enough. The false generalization is mainly caused by the second term, especially because of the high probabilities of p_{r_2} and p_{r_3} in (2) and (3), respectively.

Intuitively, this means that the negative label excessively blames the common root cause DUPath(5, 6) of the two alarms. It decreases the posterior probability of DUPath(5, 6) that also leads to decrease the probabilities of its descendants including the true alarm. Concretely, such excessive blames are incurred from unreasonably high rule probabilities that eventually lead to the false alarm. For example, the probability of a rule that derives DUPath(5, 7) should have been lower than those of other rules that derive data flow paths in straight-line code. The data flow from line 5 to 7 is involved in a loop. Since static analyses typically merge the effects of different loop iterations into a single abstract memory using join or widening operators, such data flows across loop iterations may have a higher chance of deriving false conclusions. However, the current monolithic derivation rules do not separate different circumstances.

2.4 Learning Bayesian Networks

Our main goal in this paper is to automatically derive an effective Bayesian network that minimizes the effect of false generalization. The idea is to learn more detailed features for Bayesian networks from labeled data. Figure 4 shows an overview of our learning system, BayeSmith. First, the system constructs an initial Bayesian network from the analysis results using a set of initial rules as in Figure 3. Given the Bayesian network and the bug labels, BayeSmith simulates user interactions with the alarm ranking system and observes false generalizations. Then, BayeSmith learns new rules that are likely to reduce the effect of the most serious false generalizations. If the quality of ranking improves by the new rules, then we repeat the learning process using the rules. Otherwise, BayeSmith tries another candidate rule for false generalization.

For each false generalization observed in the simulations, we find grounded rules that are the main sources of the false generalization such as $r_2(5,6,7)$ in the previous example. Then BayeSmith refines the corresponding rule based on the syntax of the program part. For example, the rule r_2 can be refined to two disjoint rules:

$$r_{21} : \text{DUPath}(c_1, c_3) \coloneq \text{DUPath}(c_1, c_2), \text{Loop}(c_2), \text{DUEdge}(c_2, c_3) \quad (5)$$

$$r_{22} : \text{DUPath}(c_1, c_3) \coloneq \text{DUPath}(c_1, c_2), !\text{Loop}(c_2), \text{DUEdge}(c_2, c_3) \quad (6)$$

where $\text{Loop}(c)$ indicates program point c corresponds to a loop head. The rule r_{21} will fire when the data flow is involved in a loop and r_{21} will fire otherwise. BayeSmith associates r_{21} with a lower probability and r_{22} with a higher one. If the new set of rules leads to improve the quality of the alarm ranking system, BayeSmith continues the learning process with the learned rules. Otherwise, it seeks to find other candidates for the false generalization or moves the focus to another false generalization in the data.

Figure 2(c) shows how the ranking of the true alarm in wget-1.12 changes whenever the user provides a label until discovering the true alarm. With the learned rules by BayeSmith, Bingo requires the user to inspect 53.4% fewer alarms by significantly reducing the magnitude and frequency of false generalizations compared to the original Bingo. The derivation graph generated by the learned rules is depicted in Figure 5. Notice that it is not achievable by only learning weights without refining rules, or uniformly refining every component in the rules. In Section 5, we will show that BayeSmith significantly outperforms these approaches. In the rest of this section, we will describe the details of the learning process.

Finding candidates. BayeSmith collects a set of candidate rules to be refined from observed false generalizations. Given a false alarm and a true alarm whose ranking is degraded by the false generalization, we first find a closest common ancestor of the alarms in the Bayesian network. For example, in Figure 2(b), the closest common ancestor of Alarm(7) and Alarm(10) is DUPath(5, 6). Then, we collect all grounded rules between the false alarm and the common ancestor. In the example, $r_2(5,6,7)$ and $r_3(5,7)$ are collected. BayeSmith tries to refine r_2 and r_3 to more precisely represent the specific circumstances at line 5, 6, and 7.

Such refinement is likely to increase the posterior probability of the true alarm. In equation (1), the conditional probability $\Pr\left(\text{Alarm}(10) \mid \neg\text{Alarm}(7)\right)$ rises if we increase the probability $\Pr\left(\text{DUPath}(5,6) \mid \neg\text{Alarm}(7)\right)$. The refinement by BayeSmith would increase this probability by decreasing the probabilities for r_2 and r_3 (see the $1 - p_{r_2}$ and $1 - p_{r_3}$ factors in equations 2 and 3).

Syntax-guided rule refinement. Next, for each candidate grounded rule, BayeSmith performs refinements guided by the syntax of the program. We assume that a program is represented as a control flow graph and each node c is associated with a command. BayeSmith is parameterized by the target language grammar for the underlying analyzer. Here we assume the following general C-like grammar:

(*Command*)	c	$\rightarrow$	$lv{:=}e \mid \texttt{assume}(e) \mid \texttt{call}(e,e) \mid \texttt{loop}$
(*L-value*)	lv	$\rightarrow$	$x \mid *e$
(*Expression*)	e	$\rightarrow$	$n \mid lv \mid e + e$

We represent the grammar in a Datalog-style representation:

	$\text{Lval}(l) \coloneq \text{Var}(l).$
$\text{Cmd}(c) \coloneq \text{Assign}(c,e)$	$\text{Lval}(l) \coloneq \text{Deref}(l,e), \text{Exp}(e).$
$\text{Cmd}(c) \coloneq \text{Assume}(c,e)$	$\text{Exp}(e) \coloneq \text{Const}(e).$
$\text{Cmd}(c) \coloneq \text{Call}(c,e)$	$\text{Exp}(e) \coloneq \text{LvalExp}(e,l), \text{Lval}(l).$
$\text{Cmd}(c) \coloneq \text{Loop}(c).$	$\text{Exp}(e) \coloneq \text{BinOp}(e,e_1,e_2), \text{Exp}(e_1), \text{Exp}(e_2).$

For all arguments of a given candidate grounded rule, BayeSmith finds the corresponding grammar rule that fires with the particular syntactic element. For example, given $r_2(5,6,7)$, BayeSmith can capture line 6 corresponds to a loop head, so that the new rule that fires with loop heads and its complement are obtained as in the rules (5) and (6). This refinement maintains the same derivability for all the output tuples. Every time a rule is refined, BayeSmith updates the rule probability to $\alpha \times p_r$ where $0 < \alpha < 1$ is a hyperparameter and p_r is the original rule probability of r. The probability for the complement rule remains unchanged.

Then BayeSmith evaluates the quality of the refined rule by running Bingo on the labeled training data. If the new rules reduce the number of user interactions in the simulation, we accept the

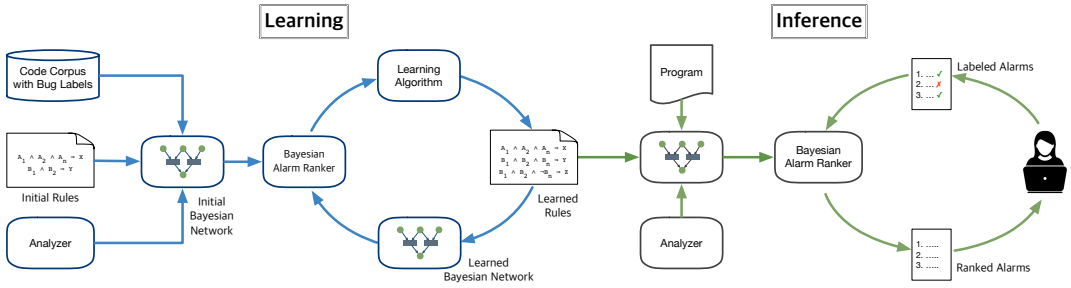


Figure 4: System overview

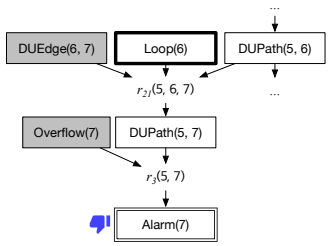


Figure 5: A portion of the learned derivation graph.

rules and repeat the same process with another false generalization if exists. When a previously refined rule is selected as a candidate again, we refine the rule further following the grammar structure. For example, the following refined rule

$$\text{DUPath}(c_1, c_3) \text{ :- } \text{DUPath}(c_1, c_2), \text{Assign}(c_2, \_), \text{DUEdge}(c_2, c_3)$$

can be further refined with the following body:

$$\text{DUPath}(c_1, c_2), \text{Assign}(c_2, e), \text{Var}(e, \_), \text{DUEdge}(c_2, c_3).$$

The rule probability then becomes $\alpha^2 \times p_r$.

In this way, BayeSmith learns rules and weights that derive Bayesian networks from a given set of training programs with bug label data. We observed that the learned models are applicable to various probabilistic program reasoning systems [5, 15, 39] and significantly reduce the user's alarm inspection burden in real world programs compared to the previous ones. We will discuss the detailed experimental results in Section 5.

3 PRELIMINARIES

3.1 Program Analysis and Syntactic Features

A Datalog program $\mathcal{D} = (\mathbf{I}, \mathbf{O}, \mathbf{R})$ is a triple of a set of input relations ($\mathbf{I}$), a set of output relations ($\mathbf{O}$), and a set of rules ($\mathbf{R}$) [1]. Each relation is a set of tuples. The output relations are derived from the input relations using the set of rules each of which is of the form $R_h(\boldsymbol{v_h}) \text{ :- } R_1(\boldsymbol{v_1}), \ldots, R_k(\boldsymbol{v_k})$ where R_h is a derived output relation and $R_1, \ldots, R_k$ are premises. We assume that the program analysis is represented as a Datalog program $\mathcal{D}_{\mathcal{A}} = (\mathbf{I}_{\mathcal{A}}, \mathbf{O}_{\mathcal{A}}, \mathbf{R}_{\mathcal{A}})$. For program analyses that are not written in a declarative language, we approximate the behavior with a Datalog program [15].

The program analysis result for program P is a tuple $\mathcal{A}(P) = (I, C, A, GC)$ where I is the set of input facts, C is the set of output facts, A is the set of alarms, and GC is the set of grounded clauses. The analysis initially sets $C := I$ and $GC := \emptyset$. Then, we accumulate the conclusions $R_h(\boldsymbol{c_h})$ and the grounded clauses $\{R_1(\boldsymbol{c_1}) \wedge \ldots \wedge R_k(\boldsymbol{c_k}) \implies_r R_h(\boldsymbol{c_h})\}$ whenever $R_1(\boldsymbol{c_1}), \ldots, R_k(\boldsymbol{c_k}) \in C$. The analysis iterates the step until reaching fixpoint.

We define a syntactic feature extractor as a Datalog program $\mathcal{D}_{\mathcal{G}} = (\mathbf{I}_{\mathcal{G}}, \mathbf{O}_{\mathcal{G}}, \mathbf{R}_{\mathcal{G}})$ that extracts relational representations of programs in grammar $\mathcal{G}$. The input relations $\mathbf{I}_{\mathcal{G}}$ consist of atomic relations such as constants and variables. The output relations $\mathbf{O}_{\mathcal{G}}$ are derived from the input relations using the grammar rules $\mathbf{R}_{\mathcal{G}}$.

3.2 Bayesian Alarm Prioritization

We present the Bayesian alarm prioritization that is a basis of various probabilistic reasoning systems [5, 15, 39]. It first extracts the derivation graph from the static analysis, converts the graph into a Bayesian network, and then computes a confidence score for each alarm. Upon user feedback, it performs Bayesian inference that updates the scores.

The results of program analysis (I, C, A, GC) form a derivation graph over the vertices $C \cup GC$. There exists an edge from a tuple $t \in C$ to a clause $g \in GC$ if t is an antecedent of g, and an edge from g to t if t is the conclusion of g.

Next, we convert the derivation graph into a Bayesian network. A Bayesian network is a tuple $(G, \mathcal{P})$ where G is a directed acyclic graph and $\mathcal{P}$ is an assignment that associates each node with a conditional probability distribution. While a Bayesian network is an acyclic graph by definition, a naive derivation graph from program analysis may have cycles. We assume that the cycle elimination algorithm in the previous work [39] is used to reduce the derivation graph to be acyclic while still preserving the derivability of all tuples. In the rest of the paper, we write G for the reduced derivation graph.

Then we assign a conditional probability for each node in the reduced graph using a given assignment $\mathcal{P}$ and rule probabilities p_r. For each input tuple t, we assign the predefined probability p_t: $\mathcal{P}(t) = p_t$. Consider a grounded clauses $g \in GC$ of the form $t_1 \wedge \ldots \wedge t_k \implies_r t_h$. Then, the conditional probability associated with g is as follows: $\mathcal{P}(g \mid t_1 \wedge \ldots \wedge t_k) = p_r$ and $\mathcal{P}(g \mid \neg(t_1 \wedge \ldots \wedge t_k)) = 0$. Consider a tuple t that is a conclusion of the grounded clauses $g_1, \ldots, g_n$. The conditional probability associated with t is as follows: $\mathcal{P}(t \mid g_1 \vee \ldots \vee g_k) = 1$ and $\mathcal{P}(t \mid \neg(g_1 \vee \ldots \vee g_k)) = 0$.

Given a set of alarms from the program analysis, the Bayesian alarm ranking system derives a list of alarms sorted in descending order of confidence. Let $\boldsymbol{e_i}$ be the set of user feedback after i iterations and Φ denote the top-ranked alarm tuple at the next iteration: $\Phi = \text{argmax}_{a \in A_u} \Pr(a \mid \boldsymbol{e_i})$ where A_u is the set of all unlabeled alarms. Given the user feedback on the top-ranked alarm Φ, we update the set $\boldsymbol{e}_{i+1}$ to $\boldsymbol{e_i} \cup \{\Phi\}$ if the user labels Φ as true, otherwise, $\boldsymbol{e_i} \cup \{\neg\Phi\}$. Then, the ranking system derives the next alarm ranking by updating the confidence scores of all remaining alarms.

4 LEARNING FRAMEWORK

4.1 Goal

The goal of our learning process is to find an optimal alarm ranking system that minimizes the number of user interactions until discovering all true alarms in the target program. We assume that a set of training programs and their bug labels are given: $\mathcal{T} = \{(P_1, L_1), \ldots, (P_N, L_N)\}$.

BayeSmith aspires to achieve the goal by learning a (finite) set of Datalog rules $\mathbf{R}^*$ that derives accurate Bayesian network models. The rules are refined by enriching the original analysis rules with syntactic features. Given a program analysis $\mathcal{D}_{\mathcal{A}} = (\mathbf{I}_{\mathcal{A}}, \mathbf{O}_{\mathcal{A}}, \mathbf{R}_{\mathcal{A}})$ and a syntactic feature extractor $\mathcal{D}_{\mathcal{G}} = (\mathbf{I}_{\mathcal{G}}, \mathbf{O}_{\mathcal{G}}, \mathbf{R}_{\mathcal{G}})$, we formalize the goal as the following optimization problem:

$$\mathbf{R}^* = \underset{\mathbf{R} \in \mathbf{R}^\infty}{\operatorname{argmin}} \sum_{(P_i, L_i) \in \mathcal{T}} \mathsf{Interaction}(P_i, L_i, \mathbf{R}) \tag{7}$$

where $\mathbf{R}^\infty = \bigcup_{n=0}^{\infty} \mathbf{R}_{\mathcal{A}} \otimes^n \mathbf{R}_{\mathcal{G}}$ and $\mathsf{Interaction}(P_i, L_i, \mathbf{R})$ is the number of user interactions until discovering all the true alarms in P_i with the ranking system derived by program analysis augmented with the syntactic features $(\mathbf{I}_{\mathcal{A}} \uplus \mathbf{I}_{\mathcal{G}} \uplus \mathbf{O}_{\mathcal{G}}, \mathbf{O}_{\mathcal{A}}, \mathbf{R})$. The set $\mathbf{R}_{\mathcal{A}} \otimes^n \mathbf{R}_{\mathcal{G}}$ is a collection of sets of analysis rules enriched with syntactic features by n times:

$$\mathbf{R}_{\mathcal{A}} \otimes^n \mathbf{R}_{\mathcal{G}} = \begin{cases} \bigcup \{\mathbf{R} \otimes \mathbf{R}_{\mathcal{G}} \mid \mathbf{R} \in (\mathbf{R}_{\mathcal{A}} \otimes^{n-1} \mathbf{R}_{\mathcal{G}})\} & n > 0 \\ \{\mathbf{R}_{\mathcal{A}}\} & n = 0 \end{cases}$$

The operator $\otimes$ refines the original analysis rules $\mathbf{R}_{\mathcal{A}}$ by adding more syntactic features from the grammar rules $\mathbf{R}_{\mathcal{G}}$. We elaborate the details of $\otimes$ in the following section.

4.2 Rule Refinement

Consider an analysis rule r and a grammar rule $r_{\mathcal{G}}$. We first define $r \otimes r_{\mathcal{G}}$ as the extensions of the analysis rule r with structure from $r_{\mathcal{G}}$. As an example, consider the rules:

r : $\mathsf{DUPath}(c_1, c_3) \coloneq \mathsf{DUPath}(c_1, c_2), \mathsf{DUEdge}(c_2, c_3).$
$r_{\mathcal{G}}$: $\mathsf{Cmd}(c) \coloneq \mathsf{Loop}(c).$

In this case, $r \otimes r_{\mathcal{G}}$ is defined as $\{r_1, r_2\}$ where

$$r_1 : \mathsf{DUPath}(c_1, c_3) \coloneq \mathsf{DUPath}(c_1, c_2), \mathsf{Loop}(c_2), \mathsf{DUEdge}(c_2, c_3). \tag{8}$$
$$r_2 : \mathsf{DUPath}(c_1, c_3) \coloneq \mathsf{DUPath}(c_1, c_2), !\mathsf{Loop}(c_2), \mathsf{DUEdge}(c_2, c_3). \tag{9}$$

The rule r_1 fires when all the conditions from the original analysis rule and the grammar rule are satisfied. The rule r_2 covers the complement of r_1 so that all the output tuples derived by the original rule r are also derived by r_1 and r_2, and vice versa. The rule probabilities of the refined rules are defined as $p_{r_1} = p_r \times \alpha$ and $p_{r_2} = p_r$ where α is a hyperparameter between 0 and 1.

We can lift the definition of $r \otimes r_{\mathcal{G}}$ to sets of analysis rules and grammar rules in a natural way: $\mathbf{R} \otimes \mathbf{R}_{\mathcal{G}}$. We postpone the formal definition of these ideas to the appendix.

The following theorem states that the refinement preserves the derivability of all output tuples from the original analysis.

Theorem 4.1 (Preservation). *Consider two Datalog programs $\mathcal{D}_1 = (\mathbf{I}_1, \mathbf{O}_1, \mathbf{R}_1)$ and $\mathcal{D}_2 = (\mathbf{I}_2, \mathbf{O}_2, \mathbf{R}_2)$. For any set I of input facts,*

$$\forall n \geq 0.\ \textit{tuple } t \textit{ is derivable from } \mathcal{D}_1 \iff t \textit{ is derivable from } \mathcal{D}$$

where $\mathcal{D} = (\mathbf{I}_1 \uplus \mathbf{I}_2 \uplus \mathbf{O}_2, \mathbf{O}_1, \mathbf{R})$ and $\mathbf{R} \in (\mathbf{R}_1 \otimes^n \mathbf{R}_2)$.

Algorithm 1: BayeSmith($\mathcal{T}, \mathcal{D}_{\mathcal{A}}, \mathcal{D}_{\mathcal{G}}$), where $\mathcal{T}$ is a set of training programs, $\mathcal{D}_{\mathcal{A}} = (\mathbf{I}_{\mathcal{A}}, \mathbf{O}_{\mathcal{A}}, \mathbf{R}_{\mathcal{A}})$ is a program analysis and $\mathcal{D}_{\mathcal{G}} = (\mathbf{I}_{\mathcal{G}}, \mathbf{O}_{\mathcal{G}}, \mathbf{R}_{\mathcal{G}})$ is a feature extractor.

```
1  Let I = I_A ∪ I_G ∪ O_G;
2  Initialize R ← R_A and ⟨Cost, FG⟩ ← Run(T, I, O_A, R_A);
3  repeat
4      for (G, a_f, a_t) ∈ FG do
5          for (r, r_G) ∈ Candidate(G, a_f, a_t) do
6              R_new ← (R \ {r}) ∪ (r ⊗ r_G);
7              ⟨Cost', FG'⟩ ← Run(T, I, O_A, R_new);
8              if Improved(Cost, Cost') then
9                  R ← R_new;
10                 ⟨Cost, FG⟩ ← ⟨Cost', FG'⟩;
11                 goto 12
12 until FG = ∅ or timeout;
13 return R;
```

4.3 Learning Algorithm

This section describes an algorithm that approximately solves the optimization problem (7) by refining rules from a set of training programs with bug labels. Ideally, the problem can be solved by maximizing true generalization and minimizing false generalization. However, it is intractable to find a global optimum because of the huge search space and non-trivial computational cost of probabilistic inference. Instead, our algorithm tries to solve the latter only using a greedy method.

The main idea is to observe false generalization as feedback and refine rules to reduce the effect. Algorithm 1 shows the refinement process of BayeSmith. Initially, the algorithm starts with the set of rules of the program analysis (line 2). Given a set of training programs $\mathcal{T}$, BayeSmith first runs the Bayesian alarm ranking system for all training programs and simulates user interactions. Then we evaluate the quality by computing the number of user interactions until discovering all the true alarms ($Cost$). The details of the algorithm will be described in the rest of this section.

BayeSmith observes all false generalizations (FG) during the simulation. For each observed false generalization, BayeSmith repeatedly refines the current rules to eliminate the false generalizations within the time budget or until there is no more false generalization (line 4). A false generalization is a tuple (G, a_f, a_t) that represents that the negative label on the false alarm a_f degrades the ranking number of the true alarm a_t in the next ranking. We define the height of a false generalization as the difference between the rankings of true alarms before and after the negative label. In our implementation, we iterate through the candidates in descending order of height.

Once a false generalization is observed, we collect candidate nodes to be refined in the Bayesian network (line 5). Given false generalization (G, a_f, a_t), we first find the lowest common ancestor t_c of a_t and a_f. Then all grounded clauses on the path from t_c to a_f in G are the refinement candidate nodes in the Bayesian network. The set of refinement candidates $\mathsf{Candidate}(G, a_f, a_t)$ is formally defined as follows:

$$\{(r, r_{\mathcal{G}}) \in \mathbf{R} \times \mathbf{R}_{\mathcal{G}} \mid gc \text{ is on the path from } t_c \text{ to } a_f \text{ in } G \text{ and } \exists 0 \leq k \leq N. R_k(c_k) \text{ is derived by } \mathcal{D}_{\mathcal{G}} \text{ using rule } r_{\mathcal{G}}\}$$

Table 1: Benchmark characteristics.

Program	KLOC	#Bugs	Bug Type	Reference
gzip-1.2.4a	9	14	Buffer overrun	[14]
fribidi-1.0.7	13	1	Buffer overrun	[34]
bc-1.06	14	2	Buffer overrun	[14]
cflow-1.5	40	1	Buffer overrun	[33]
patch-2.7.1	51	1	Buffer overrun	[27]
wget-1.12	65	6	Buffer overrun	[41, 42]
readelf-2.24	65	1	Buffer overrun	[31]
grep-2.19	68	1	Buffer overrun	[25]
sed-4.3	83	1	Buffer overrun	[9]
sort-7.2	98	1	Buffer overrun	[7]
tar-1.28	112	1	Buffer overrun	[24]
jhead-3.0.0	5	2	Integer overflow	[32]
shntool-3.0.5	13	6	Integer overflow	[14]
autotrace-0.31.1	18	18	Integer overflow	[30]
sam2p-0.49.4	22	12	Integer overflow	[28]
sdop-0.61	23	65	Format string	[31]
latex2rtf-2.1.1	27	2	Integer overflow	[29]
urjtag-0.8	46	6	Format string	[26]
optipng-0.5.3	61	1	Integer overflow	[14]
a2ps-4.14	64	6	Format string	[14]

where $gc =$ " $\bigwedge_{i=0}^{N} R_i(c_i) \implies_r R_h(c_h)$" and $R_k \in \mathbf{O}_{\mathcal{G}}$.

For each candidate, BayeSmith refines the selected rule using the refinement operator defined in Section 4.2 (line 6). The quality of the rules are examined by simulating user interaction with the newly learned alarm ranking system (line 7). For evaluation, we check (line 8) (1) whether the total number of user interactions for all training programs decreases, and (2) whether the number of improved cases is larger than that of hindered cases. While the first condition generally specifies the ultimate goal, the second one is checked to avoid overfitting by a few dominating programs that have a large number of alarms. If the conditions are satisfied, BayeSmith repeats the process with the newly learned rules. Otherwise, other candidates of rules or false generalization are selected.

Example 4.2. Consider the Bayesian network in Figure 2(b). Given the false alarm $\mathsf{Alarm}(7)$ and the true alarm $\mathsf{Alarm}(10)$, the lowest common ancestor is $\mathsf{DUPath}(5,6)$. BayeSmith collects all grounded clauses on the path from $\mathsf{DUPath}(5,6)$ to $\mathsf{Alarm}(7)$: $\{r_2(5,6,7), r_3(5,7)\}$. Then the algorithm searches for a grammar rule that derives a tuple whose argument is 5, 6 or 7. If there exists a rule $r_{\mathcal{G}} : \mathsf{Cmd}(c) \coloneqq \mathsf{Loop}(c)$ that derives $\mathsf{Cmd}(6)$, the pair $(r_2, r_{\mathcal{G}})$ is selected as a candidate. In this case, the algorithm refines the rule and produces two new rules (8) and (9) in Section 4.2. Tables **??**–**??** in Appendix **??** provide additional examples of refined rules learned by BayeSmith.

5 EXPERIMENTAL EVALUATION

We designed experiments to answer the following questions:

(1) How effective are the learned probabilistic models?
(2) Does BayeSmith reduce the frequency and magnitude of false generalizations?
(3) How robust is BayeSmith with different training data?
(4) How scalable are the learned ranking systems?

5.1 Setting

We conducted all experiments on Linux machines with Intel Xeon 2.2GHz. We performed Bayesian inference using libDAI [35] and set a timeout of 24 hours for learning.

Instance analyses. We have implemented BayeSmith on top of Sparrow, a static analysis framework for C programs [37]. We used two instance analyses in Sparrow: an interval analysis for buffer-overrun errors, and a taint analysis for format-string and integer-overflow errors. The taint analysis detects whether malicious format strings and overflowed integers are used as arguments of `printf`-like and `malloc`-like functions, respectively. We used the same mechanism as in the previous work [5, 15] to extract def-use relations from analysis results of Sparrow following the sparse analysis framework [36]. For the grammar rules, we used the C-like grammar (as in Section 2) that describes program syntax and alarm expressions in Sparrow.

Baselines. We apply probabilistic models learned by BayeSmith for three state-of-the-art probabilistic program reasoning systems each of which prioritizes alarms based on the feedback from user [39], the old version of the program [15], and dynamic analysis [5], respectively. All three baselines are based on probabilistic models derived from the same set of hand-written rules. We compare the performance of the learned models for each system.

Benchmarks. We evaluated BayeSmith on a suite of widely used C programs in Table 1. The benchmarks are collected from previous work applying Sparrow [5, 14, 15] and recent CVE reports. We excluded too small programs whose sizes are less than 5KLOC and alarm inspection requires less than 5 user interactions with Bingo.

Learning configuration. By default, we evaluated the performance of BayeSmith using the leave-one-out cross-validation for each instance analysis and set the hyper-parameter α in Section 4.2 to 0.99. Our leave-one-out cross-validation measures the number of user interactions for each individual program using a Bayesian network learned from the other programs. For example, we used 10 programs for training and the remaining one program for the test, in the case of the interval analysis. Because each analysis is based on the different abstract domains and semantics, we separated the learning processes per analysis.

5.2 Effectiveness of Probabilistic Models

5.2.1 User-guided Alarm Prioritization. We evaluate the effectiveness of the learned models for user-guided alarm prioritization compared to three baselines: Bingo_M, Bingo_{EM}, and Bingo_U. Bingo_M derives Bayesian networks using the initial set of rules and the rule probabilities are assigned with a heuristically chosen value (0.99) as in the previous work [15]. Bingo_{EM} constructs the same Bayesian networks as Bingo_M but the rule weights are learned by an algorithm presented in the previous work [39]. The algorithm learns the best rule probabilities that explain the bug labels based on the standard EM algorithm. We ran Bingo_{EM} five times for each benchmark and report the average. Bingo_U uses a refined set of rules that are derived by uniformly unrolling all the components of the original set of rules by once. We measure the number of user interactions to discover all true bugs in the data. The experimental results are shown in Table 2.

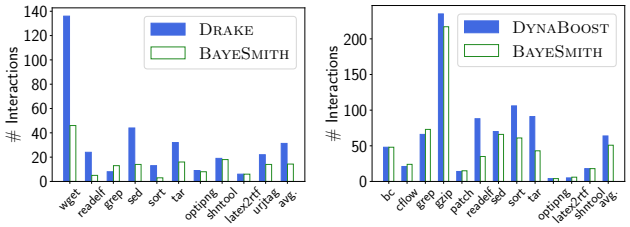
Table 2: Effectiveness of BayeSmith in user-guided alarm prioritization for the interval (top) and taint (bottom) analysis. #Alarms reports the number of alarms. Each column reports the number of iterations until discovering all bugs.

Program	#Alarms	Bingo$_M$	Bingo$_{EM}$	Bingo$_U$	BayeSmith
gzip-1.2.4a	358	145	325	188	107
fribidi-1.0.7	213	6	2	5	3
bc-1.06	535	96	100	97	94
cflow-1.5	805	94	94	91	60
patch-2.7.1	502	36	47	30	34
wget-1.12	891	193	138	179	90
readelf-2.24	882	78	25	81	18
grep-2.19	912	53	286	59	53
sed-4.3	819	122	144	127	121
sort-7.2	715	176	160	178	94
tar-1.28	1,369	218	308	205	146
Total	8,001	1,217	1,628	1,240	820
jhead-3.0.0	19	7	12	7	4
shntool-3.0.5	23	14	19	15	10
autotrace-0.31.1	77	77	71	77	43
sam2p-0.49.4	20	20	20	20	20
sdop-0.61	150	85	81	81	81
latex2rtf-2.1.1	13	6	9	6	6
urjtag-0.8	35	22	24	22	17
optipng-0.5.3	67	14	16	13	12
a2ps-4.14	27	15	13	14	11
Total	431	260	264	255	204

The weight learning performed by Bingo$_{EM}$ is not effective in most cases, and only improves the ranking quality for 7 out of 20 programs compared to Bingo$_M$. For some cases such as grep-2.19 and gzip-1.2.4a, the number of interactions even significantly increases. This result is mainly because of two reasons. First, the labeling is extremely sparse in our setting. For example, we only have tens of bug labels for the interval analysis while the networks have 2K–12K nodes. This hinders the converge of the algorithm within 12 hours that leads to poor performance. The other reason is that the networks have simplistic structures derived by the small and monolithic set of rules, that hinders to learn general knowledge for detailed contexts.

The results also show that the uniform rule refinement does not improve the quality of the ranking system. In the interval analysis, Bingo$_U$ outperforms Bingo$_M$ for 5 benchmarks but increases the number of interactions by 1.9% for all benchmarks, on average. Similarly, Bingo$_U$ improves only 1.9% in the taint analysis compared to Bingo$_M$. Overall, the uniformly unrolled rule does not effectively solve the false generalization problem according to our results. This is mainly because the unguided refinement can reduce not only false generalizations but also true generalizations, thereby degrading the overall performance. Furthermore, one-step refinements are sometimes not enough to remove false generalizations.

On the other hand, the learned ranking systems substantially reduce the amount of alarm inspection burden. BayeSmith outperforms Bingo$_M$ for 17 out of 20 programs, and reduces the total number of user interactions by 32.6% and 21.5% for the interval and taint analysis, respectively. For the remaining 3 cases, BayeSmith



(a) Drake, 54.3% ↓ (b) DynaBoost, 20.4% ↓

Figure 6: Improvement in ranking performance enabled by BayeSmith

shows the same results compared to Bingo$_M$. This is mainly because the underlying analyzer already reports a low false positive ratio (e.g., latex2rtf-2.1.1) or Bingo$_M$ already significantly reduces alarm inspection burden (e.g., grep-2.19).

5.2.2 Application to Other Systems. We evaluate the effectiveness of learned models with two other probabilistic reasoning systems using Bayesian networks: Drake and DynaBoost. Drake bootstraps the probabilistic alarm ranking system using the old version of the program and prioritizes alarms for the new version by the relevance to the difference [15]. DynaBoost incorporates observed dataflow facts from dynamic analysis into the alarm ranking system [5]. As in the prior work, both of the baselines use the same set of rules as Bingo$_M$ in the previous section. Likewise, We measure the number of interactions after each bootstrapping until discovering all the bugs. Since they require old versions and test cases, we selected benchmark programs only used in the previous work.

Overall, the learned models significantly improve the performance of the probabilistic reasoning systems. Figure 6(a) shows the number of user interactions on top of the alarm ranking by the relevance. In comparison to the vanilla versions of Drake, the alarm inspection burden with the learned models is reduced by 54.3%, on average. One exceptional case is grep-2.19 where the baseline already significantly reduces the number of interactions to only 8 to discover the bug. While the learned rules show a small regression, the absolute number of user interactions is still small enough (13) for the user inspection. Figure 6(b) also demonstrates the impact on the ranking system incorporating dynamic analysis. The learned models reduce the number of interactions for 8 out of 12 programs for DynaBoost and the user inspection burden is reduced by 20.4%, on average.

5.2.3 Learned Insights. The learned rules capture important aspects of each analysis. Regardless of the instance analysis, BayeSmith commonly captures the insight that loop head is an important feature since loops are often the main source of inaccuracy because of join or widening. Another insight is that library calls also increase uncertainty of static analysis results because their semantics is complicated and the source code is unavailable. BayeSmith not only discovers these general insights in static analysis, but also fine-tunes the rules with multiple conjunctions and disjunctions, which can require non-trivial manual efforts[3]:

$$\mathsf{DUPath}(v_0, v_1) \coloneq \ldots, \mathsf{Loop}(v_2), \mathsf{Assign}(v_2, \_). \quad : 0.97$$
$$\mathsf{DUPath}(v_0, v_1) \coloneq \ldots, \mathsf{LibCall}(v_2, v_3), \mathsf{Lval}(v_3, \_). \quad : 0.97$$

[3]For brevity, we omit negated tuples and boilerplate parts in the rule bodies.

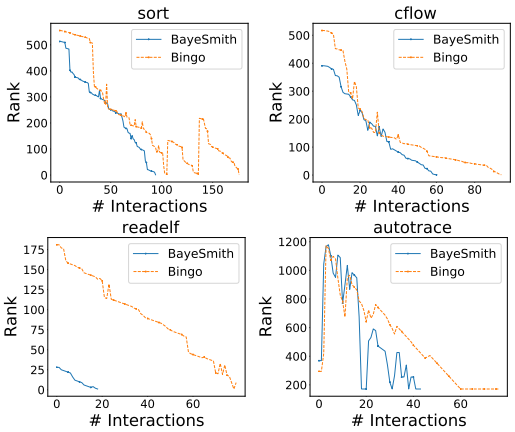


Figure 7: Comparison of the ranking performance. Each data point represents the sum of the rankings of all bugs.

BayeSmith also captures analysis-specific features. For the interval analysis, it is usually harder to estimate buffer overflow errors when arithmetic operators are involved in branch conditions or buffer accesses. Also the learned rules capture the general knowledge that analyzing pointer dereferences or complicated libraries such as `strncpy` is more tricky compared to array index expressions. These insights are captured by the following rules and weights:

$$\begin{aligned}
\mathsf{DUPath}(v_0, v_1) &:\!- \ldots, \mathsf{Assume}(v_2, v_3), \mathsf{BinOp}(v_3, _, _, _). && : 0.97\\
\mathsf{Alarm}(v_1) &:\!- \ldots, \mathsf{DerefExp}(v_1, v_2), \mathsf{BinOp}(v_2, _, _). && : 0.97\\
\mathsf{Alarm}(v_1) &:\!- \ldots, \mathsf{Strncpy}(v_1, v_2, v_3), \mathsf{Cast}(v_2, _), \mathsf{Cast}(v_3, _). && : 0.96
\end{aligned}$$

For the taint analysis, arithmetic operators such as multiplication and casting operators are captured by BayeSmith, which are the main sources of imprecision for integer overflow detection:

$$\begin{aligned}
\mathsf{Alarm}(v_1) &:\!- \ldots, \mathsf{MallocSize}(v_1, v_2), \mathsf{CastExp}(v_2, v_3), \mathsf{Lval}(v_3, _). && : 0.96\\
\mathsf{Alarm}(v_1) &:\!- \ldots, \mathsf{MallocSize}(v_1, v_2), \mathsf{BinOp}(v_2, v_3, _, _), \mathsf{Mult}(v_3). && : 0.96
\end{aligned}$$

5.3 Reduction of False Generalization

BayeSmith achieves its effectiveness by reducing false generalizations. We justify this argument by measuring the frequency and magnitude of false generalizations. Table 3 shows the number of false generalizations and their average magnitudes by Bingo$_M$ and BayeSmith. We measured the magnitude of false generalization in each round as the sum of decreased amount in ranking per true alarm. The numbers are the average throughout the whole interaction rounds. The overall changes in the ranks of true alarms are shown in Figure 7. Each graph describes the rankings of true alarms for each iteration.

We categorize the results into three notable cases. First, in most of the cases, BayeSmith reduces the number of false generalizations and magnitudes and leads to the performance improvement of their rankings such as sort-7.2 and cflow-1.5. Second, the learned Bayesian networks improve the quality of ranking systems in terms of not only generalization but also initial ranking. For example, the initial ranking of the bug in readelf-2.24 is improved from 181 to 28, and hence the reduced number of total interactions. This means that the learned probabilistic models estimate the behavior

Table 3: Reduction of false generalizations. Freq and Mag report the number of false generalizations and their average magnitude for the interval (top) and taint (bottom) analysis.

	Bingo$_M$			BayeSmith		
Program	**Freq**	**Mag**	**Freq×Mag**	**Freq**	**Mag**	**Freq×Mag**
gzip-1.2.4a	191	15.4	2931.9	130	16.0	2080.0
fribidi-1.0.7	2	2.0	4.0	0	0.0	0.0
bc-1.06	11	121.6	1337.9	7	55.3	387.0
cflow-1.5	7	22.1	155.0	5	21.6	108.0
patch-2.7.1	4	26.0	104.0	3	1.3	3.9
wget-1.12	125	86.7	10842.5	40	59.7	2388.0
readelf-2.24	6	8.8	53.0	1	1.0	1.0
grep-2.19	0	0.0	0.0	0	0.0	0.0
sed-4.3	5	51.0	255.0	8	57.1	457.0
sort-7.2	7	73.0	511.0	3	13.0	39.0
tar-1.28	27	19.4	523.0	26	7.8	201.8
Average	35	38.7	1355.6	20	21.2	429.0
jhead-3.0.0	0	0.0	0.0	0	0.0	0.0
shntool-3.0.5	9	1.0	9.0	10	1.3	13.0
autotrace-0.31.1	194	12.1	2341.6	251	11.7	2941.7
sam2p-0.49.4	79	3.5	279.7	73	3.8	276.7
sdop-0.61	1552	5.5	8582.6	1834	3.3	6015.5
latex2rtf-2.1.1	0	0.0	0.0	0	0.0	0.0
urjtag-0.8	6	11.0	66.0	0	0.0	0.0
optipng-0.5.3	3	21.7	65.0	3	12.7	38.0
a2ps-4.14	8	4.1	33.0	6	3.2	19.0
Average	206	6.5	1346.5	242	4.0	965.1

of programs and static analyses with more accurate prior probabilities. Third, the learned ranking systems sometimes report more false generalizations such as autotrace-0.31.1. However, notice that Bingo$_M$ exhaustively inspects all the alarms to discover the true alarms. The rankings of the true alarms are consistently low and all the 18 bugs are discovered in the last 18 iterations, hence less number of false generalizations. On the other hand, the first true alarm is discovered only after 18 iterations with BayeSmith that improves the rankings of the other true alarms. This effect makes a high chance of introducing false generalizations, but significantly improves the overall rankings.

In summary, the learned rules by BayeSmith reduce the negative effect by false generalizations. For the interval analysis, BayeSmith reduces the average frequency and magnitude of false generalizations by 42.9% and 45.2%, respectively. For the taint analysis, the average magnitude is reduced by 38.5%. The average number of false generalizations increases because of the exceptional cases, but even for such cases, the quality of overall ranking improves. In terms of overall impact (**Freq** × **Mag**), BayeSmith shows 68.4% and 28.3% of improvement, respectively.

5.4 Robustness of the Learning Algorithm

In order to measure how data splitting affects the quality of learned models, we evaluate the performance of BayeSmith with different quantities of training and testing data. For each analysis, we randomly chose 80% of the benchmark programs as a training set and the remaining 20% as a test set. We repeated this process ten times and report the averages. We measure the total number of

Table 4: Statistics of user interactions using the alarm ranking systems learned with different training set.

	BayeSmith$_{90}$		BayeSmith$_{80}$	
Program	**Avg.**	**Stddev.**	**Avg.**	**Stddev.**
Interval	65.0	40.7	65.2	41.0
Taint	18.8	23.0	19.0	22.9

iterations for each test set (BayeSmith$_{80}$) compared to that of the leave-one-out setting that is the same as Section 5.2 (BayeSmith$_{90}$).

Table 4 shows the performance of BayeSmith with 10 different combinations of training and test sets. On average, BayeSmith$_{80}$ reduces the number of user interactions by 34.3% for the interval analysis and 18.8% for the taint analysis. Compared to BayeSmith$_{90}$, BayeSmith$_{80}$ shows 0.2% and 1.0% less improvement for each of the analysis task. In total, the results are comparable to those with BayeSmith$_{90}$ in Section 5.2 that shows the learning algorithm is robust with different training data.

5.5 Scalability

The computational overhead of Bayesian inference typically increases as the size of Bayesian network grows because of the refinements by BayeSmith. We measure the size of Bayesian networks in terms of the number of tuples and grounded clauses in derivation trees and the average running time for each Bayesian inference. The Bayesian networks are optimized using optimization algorithms described in the previous work [39].

Table 5 shows the average size and response time. Bingo uses a monolithic set of 7 rules while BayeSmith derives 25 and 13 rules on average for each analysis. On average, the synthesized networks have 1.4x more nodes for the interval analysis and 1.1x for nodes for the taint analysis, compared to the baseline. As the networks become larger, the average running time also increases by 5.1x and 2.6x, respectively. In most cases, it only requires less than 10 seconds. Even for some exceptional cases such as readelf-2.24, the average response time is about a minute, which is reasonable for real world deployment.

6 LIMITATIONS AND OPPORTUNITIES

This section identifies the limitations of our approach and challenges for future work. First, BayeSmith requires background knowledge in the form of existing analysis rules. We demonstrated a set of rules for def-use relations based on the sparse analysis framework [36] is a reasonable starting point. While def-use relations are used in a large class of analyses [16, 44, 49], there can be other analyses that require the analysis designers to derive initial rules based on different background knowledge. To address this challenge, we plan to devise a purely data-driven synthesis algorithm without requiring background knowledge. Second, BayeSmith only considers a syntax-guided refinement that might not accurately reflect deeper semantic aspects. For future work, we plan to develop a refinement algorithm considering the characteristics of abstract domains and semantics. Finally, BayeSmith assumes an ideal user interaction model where the user always gives correct feedback on the top-ranked alarm. However, the user might make a mistake, give partial feedback, or diagnose arbitrary alarms. Thus, developing a more advanced interaction model is also an important future direction for our work.

7 RELATED WORK

Our approach aims to overcome the fundamental limitation of the existing user-guided approaches. There have been several techniques to improve static analysis accuracy by leveraging user feedback. Bayesian alarm ranking systems compute a confidence score for each alarm and update rankings based on the user feedback [15, 39]. Alarm clustering computes logical correlations between alarms so that a set of alarms are logically grouped together [21, 22], such that each alarm group is resolved by inspecting a single representative alarm. Alarm classification techniques ask questions about labels or root causes of alarms to classify analysis reports by solving optimization problems [23, 50]. None of the existing approaches learn detailed knowledge from an analysis for one program to that for other programs. Instead, we provide a learning framework for probabilistic models and outperform the existing approaches.

Recently, many techniques for synthesizing Datalog programs have been proposed [3, 40, 45, 46], but their goal is different from BayeSmith. Their goal is to efficiently synthesize Datalog rules from input-output specifications. Zaatar [3] proposes a constraint-based synthesis by encoding the output of candidate Datalog programs as SMT constraints. Alps [45] leverages a search space pruning technique for efficient enumerative synthesis. Difflog and ProSynth employ provenance information combined with continuous optimization [46] and SAT solving [40]. One notable difference is that Datalog programs are typically interpreted over a Boolean semiring, where each tuple is either derived or not, while our focus in alarm prioritization is to rank alarms in order of confidence, so each tuple is additionally associated with a real-valued score.

Existing structure learning techniques that have been developed by the machine learning community [11, 43, 48] are not applicable to our problem. The goal of structure learning algorithms for Bayesian networks and Markov Logic Networks has been to find a model with a high likelihood on the observed data. In contrast, BayeSmith finds a refinement of an existing model, which is tightly coupled with the underlying static analysis, while preserving the derivability of the original set of alarms (Theorem 4.1). Furthermore, conventional structure learning algorithms have limited scalability since they typically learn rules from scratch. The standard approaches to structure learning typically require complete data (i.e., labels for all random variables). However, this assumption is unrealistic for our problem since it is hard to obtain labels for all intermediate results of static analysis. Although several techniques [2, 8] can learn structures from incomplete data, they are only applicable to small datasets with up to a few dozen random variables, while our benchmarks, on average, have more than 1K variables (Table 5).

There is a large body of research for automatically learning heuristics for static analysis [4, 10, 12–14, 17–19, 38, 47]. The existing approaches use various learning techniques to derive heuristics for controlling context-sensitivity [17, 19], variable relationship [10, 12, 47], resource management [13], and unsoundness [14]. While all these approaches rely on handcrafted features, our approach learns syntactic features that are directly derived from the grammar of the target language. Jeon et al. [18] automatically learns

Table 5: Scalability of our learning approach. #T and #C are the average number of tuples and clauses of the networks. #R indicates the number of rules to construct a network. Time is the average duration of Bayesian inference.

	Bingo			BayeSmith			
Program	**#T**	**#C**	**Time (s)**	**#R**	**#T**	**#C**	**Time (s)**
Interval	3,053	3,385	2.9	25	4,414	4,746	14.9
Taint	422	541	1.2	13	475	595	3.0

graph-based heuristics without manual features, but their approach is specialized for pointer analysis. Chae et al. [4] automatically generates features from program syntax. They represent features as small programs that concisely capture the behavior of static analysis with flow-sensitivity or relational domain. However, these approaches are not directly applicable to learning Bayesian networks for alarm ranking systems.

8 CONCLUSION

We presented BayeSmith, a general learning framework for Bayesian alarm ranking systems. BayeSmith fundamentally improves the quality of user-guided program reasoning systems by learning the structure of the underlying probabilistic model. BayeSmith learns accurate Bayesian networks by minimizing the effect of false generalizations observed from simulated user interactions with labeled data. In the experiments with two instance analyses, we demonstrated the effectiveness of learned rules and significant improvement in terms of the user's alarm inspection burden.

ACKNOWLEDGMENTS

This work was partly supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944, 2021R1C1C1003876) and Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2021-0-00758, Development of Automated Program Repair Technology by Combining Code Analysis and Mining). Raghothaman was supported by U.S. NSF Grant CCF #2107261.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1994. *Foundations of Databases: The Logical Level* (1st ed.). Pearson.
[2] Tameem Adel and Cassio P. de Campos. 2017. Learning Bayesian Networks with Incomplete Data by Augmentation. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI Press, 1684–1690.
[3] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017*, Vol. 10416. Springer, 689–706.
[4] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically generating features for learning program analysis heuristics for C-like languages. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 101:1–101:25.
[5] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting Static Analysis Accuracy with Instrumented Test Executions. In *Proceedings of theThe ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE 2021)*.
[6] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *J. Log. Comput.* (1992).
[7] Paul Eggert. 2010. `sort`: Commit `14ad7a2`. http://git.savannah.gnu.org/cgit/coreutils.git/commit/?id=14ad7a2. `sort`: Fix very-unlikely buffer overrun when merging to input file.
[8] Nir Friedman. 1998. The Bayesian Structural EM Algorithm. In *UAI '98: Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*. 129–138.
[9] Assaf Gordon. 2018. `sed`: Commit `007a417`. http://git.savannah.gnu.org/cgit/sed.git/commit/?id=007a417. `sed`: Fix heap buffer overflow from multiline EOL regex optimization.
[10] Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2020. Learning fast and precise numerical analysis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*. ACM, 1112–1127.
[11] David Heckerman, Dan Geiger, and David Maxwell Chickering. 1995. Learning Bayesian Networks: The Combination of Knowledge and Statistical Data. *Mach. Learn.* 20, 3 (1995), 197–243.
[12] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis. In *Static Analysis - 23rd International Symposium, SAS (Lecture Notes in Computer Science, Vol. 9837)*. Springer, 237–256.
[13] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-aware program analysis via online abstraction coarsening. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. IEEE / ACM, 94–104.
[14] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided Selectively Unsound Static Analysis. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*. IEEE Press, 519–529.
[15] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. 2019. Continuously reasoning about programs using differential Bayesian inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, 561–575.
[16] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis*. Springer, 238–255.
[17] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 140:1–140:29.
[18] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 179 (2020).
[19] Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 100:1–100:28.
[20] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: Principles and techniques*. The MIT Press.
[21] Wei Le and Mary Lou Soffa. 2010. Path-based fault correlations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010)*. ACM, 307–316.
[22] Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Sound Non-Statistical Clustering of Static Analysis Alarms. *ACM Trans. Program. Lang. Syst.* 39, 4 (2017), 16:1–16:35.
[23] Ravi Mangal, Xin Zhang, Aditya Nori, and Mayur Naik. 2015. A user-guided approach to program analysis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 462–473.
[24] Jim Meyering. 2018. `tar`: Commit `b531801`. http://git.savannah.gnu.org/cgit/tar.git/commit/?id=b531801. One-top-level: Avoid a heap-buffer-overflow.
[25] MITRE. 2015. CVE-2015-1345. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1345.
[26] MITRE. 2015. CVE-2015-8106. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8106.
[27] MITRE. 2016. CVE-2016-10713. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10713.
[28] MITRE. 2017. CVE-2017-16663. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16663.
[29] MITRE. 2017. CVE-2017-16938. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16938.
[30] MITRE. 2017. CVE-2017-9181. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9181.
[31] MITRE. 2018. CVE-2018-10372. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10372.
[32] MITRE. 2018. CVE-2018-6612. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6612.
[33] MITRE. 2019. CVE-2019-16166. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16166.
[34] MITRE. 2019. CVE-2019-18397. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18397.
[35] Joris Mooij. 2010. libDAI: A Free and Open Source C++ Library for Discrete Approximate Inference in Graphical Models. *Journal of Machine Learning Research* 11 (Aug. 2010), 2169–2173.
[36] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and Implementation of Sparse Global Analyses for C-like Languages. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2012)*. ACM, 229–238.
[37] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. The Sparrow static analyzer. https://github.com/ropas/sparrow.

[38] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 572–588.

[39] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided Program Reasoning Using Bayesian Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 722–735.

[40] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2020. Provenance-guided synthesis of Datalog programs. *Proc. ACM Program. Lang.* 4, POPL (2020), 62:1–62:27.

[41] Tim Rühsen. 2018. `wget`: Commit `b3ff8ce`. http://git.savannah.gnu.org/cgit/wget.git/commit/?id=b3ff8ce. `src/ftp-ls.c` (`ftp_parse_vms_ls`): Fix heap-buffer-overflow.

[42] Tim Rühsen. 2018. `wget`: Commit `f0d715b`. http://git.savannah.gnu.org/cgit/wget.git/commit/?id=f0d715b. `src/ftp-ls.c` (`ftp_parse_vms_ls`): Fix heap-buffer-overflow.

[43] Mauro Scanagatta, Cassio P. de Campos, Giorgio Corani, and Marco Zaffalon. 2015. Learning Bayesian Networks with Thousands of Variables. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015*. 1864–1872.

[44] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 693–706.

[45] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-guided synthesis of Datalog programs. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018*. ACM, 515–527.

[46] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*. ijcai.org, 6117–6124.

[47] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *Computer Aided Verification - 30th International Conference, CAV (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 211–229.

[48] Peter Spirtes, Clark Glymour, and Richard Scheines. 2000. *Causation, Prediction, and Search, Second Edition*. MIT Press.

[49] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, 265–266.

[50] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 57 (Oct. 2017), 30 pages.